Evaluating automatically parallelized versions of the support vector machine

Codreanu, Valeriu; Droge, Bob; Williams, David; Yasar, Burhan; Yang, Fo; Liu, Baoquan; Dong, Feng; Surinta, Olarik; Schomaker, Lambertus; Roerdink, Jos

Published in:
Concurrency and Computation

DOI:
10.1002/cpe.3413

Publication date:
2014

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Evaluating automatically parallelized versions of the Support Vector Machine

Valeriu Codreanu[1,6*], Bob Dröge[2], David Williams[1], Burhan Yasar[5], Po Yang[4], Baoquan Liu[4], Feng Dong[4], Olarik Surinta[3], Lambert R.B. Schomaker[3], Jos B.T.M. Roerdink[1], and Marco A. Wiering[3]

[1]*University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science, The Netherlands*
[2]*University of Groningen, Donald Smits Centrum voor Informatie Technologie, The Netherlands*
[3]*University of Groningen, Institute of Artificial Intelligence and Cognitive Engineering, The Netherlands*
[4]*University of Bedfordshire, Department of Computer Science and Technology, United Kingdom*
[6]*Eindhoven University of Technology, Electronic Systems Group, Eindhoven, The Netherlands*
[5]*Rotasoft Inc., Ankara, Turkey*

SUMMARY

The Support Vector Machine (SVM) is a supervised learning algorithm used for recognizing patterns in data. It is a very popular technique in Machine Learning and has been successfully used in applications such as image classification, protein classification, and handwriting recognition. However, the computational complexity of the kernelized version of the algorithm grows quadratically with the number of training examples. To tackle this high computational complexity we have developed a directive-based approach that converts a gradient-ascent based training algorithm for the CPU to an efficient GPU implementation. We compare our GPU-based SVM training algorithm to the standard LibSVM CPU implementation, a highly-optimized GPU-LIBSVM implementation, as well as to a directive-based OpenACC implementation. The results on different handwritten digit classification datasets demonstrate an important speed-up for the current approach when compared to the CPU and OpenACC versions. Furthermore, our solution is almost as fast and sometimes even faster than the highly optimized CUBLAS-based GPU-LIBSVM implementation, without sacrificing the algorithm's accuracy. Copyright © 2014 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: GPU, Automatic Parallelization; Handwritten Digit Recognition; Machine Learning, Support Vector Machine

1. INTRODUCTION

The massive Internet growth coupled with the rise of mobile devices capable of generating huge amounts of information has led in recent years to the phenomenon of *Big Data*. Every day, 2.5 quintillion bytes (exabytes) of data are created. This translates to more than 90% of today's data being created in the last two years alone [4]. However, extracting meaningful knowledge from this ever-increasing *sea of information* is a very challenging task. In this context, Machine Learning (ML) promises to offer the necessary tools for dealing with these data in the most intelligent way.

There are several open-source machine learning toolkits that provide support for developing ML applications [13, 35, 37], but in cases where datasets exceed the size and complexity of trivial problems, these algorithms tend to be very computationally expensive. Machine learning algorithms are heavily based on linear algebra operations such as matrix-matrix multiplications or

*Correspondence to: Den Dolech 2, Eindhoven, The Netherlands, E-mail: v.codreanu@tue.nl

dot products between vectors. The data on which these operations are applied are often independent, making batch ML algorithms amenable to execution on parallel hardware systems, such as Graphics Processing Units (GPUs).

Moore's law is still governing the semiconductor industry and it applies to all types of integrated circuits, and hence also to GPUs. Each new hardware generation increases the GPU-CPU performance gap, especially with regards to single-precision floating point operations [29]. This can also be observed in Figure 1. When considering the GFLOPs/$ metric, the ratio between high-end GPUs and CPUs becomes even greater [12]. Moreover, with the integration of CPUs and GPUs on the same die, communication costs between the two devices are reduced and the potential for acceleration is much higher.

Besides the fact that GPUs offer higher computational throughput than CPUs, programming them has also become much easier. Advances in specialized libraries and compiler tools such as the one presented in this paper bring GPU computing closer to the typical scientist who does not necessarily have a High Performance Computing (HPC) background.

In this paper we strive to provide a fair comparison between several parallel implementations of the Support Vector Machine [54]. Existing libraries and semi-automatic parallelization tools are used, and both accuracy and execution times are presented. We have developed a gradient ascent SVM training algorithm, which is very suitable for automatic parallelization. In contrast to other SVM optimization techniques, the gradient ascent algorithm can work on all data in parallel, allowing for a large computational performance gain when executed on GPU devices.

Our experiments use handwritten digit datasets. The goal is to infer the correct label of the images. This is done by training the SVM using a training dataset and then evaluating the obtained classifier on a test dataset, as is usual in machine learning. We have used the classic dataset for the digit classification problem, the MNIST dataset [30], and additionally a dataset containing Bengali handwritten digits. Furthermore, we examine the utility of two novel feature extraction methods, which convert the pixel intensity images to high-dimensional feature vectors.

Novel contributions. This paper describes a novel approach for parallelizing a training algorithm for the support vector machine. Because we constructed a gradient ascent technique to optimize the variables in the support vector machine, examples are treated independently of each other. Therefore, the parallelization of our algorithm is more straightforward and efficient than is the case with previous techniques. Furthermore, we perform a number of different experiments on two challenging handwritten digit datasets. To obtain more accurate results we have invented two novel feature extraction techniques which convert the pixel values in a handwritten image to a high-dimensional input vector. Finally, we have performed extensive comparisons between our GPU-SVM algorithm and existing state-of-the-art methods. The results of these comparisons show that our method is the fastest of all evaluated methods if the examples consist of high-dimensional feature vectors. By exploiting the speed of our method, we were also able to better tune the meta-parameters of the SVM, leading to the highest accuracy so far on the Bangla dataset.

Outline of the paper. Section 2 presents the context of this work and the relevant background. Section 3 gives a formal description of the SVM and the learning techniques used. Section 4 presents the automatic parallelization tools used for the evaluation, as well as some details on the implementations. Section 5 reviews the datasets used for the experiments and the feature extraction methods that will be compared. Section 6 presents the results. Finally, in Section 7 we draw the conclusions and outline future directions.

2. BACKGROUND

2.1. Machine learning on GPUs

GPU technology has evolved steadily over the last 10 years. Initially, GPUs were only used for rendering graphics, but in recent years a shift to the general-purpose use of GPUs (GPGPU) is clearly observed. This stems from the highly-parallel structure of the GPU, which offers great potential for acceleration of several classes of algorithms. GPU computing was first adopted by

DOI: 10.1002/cpe

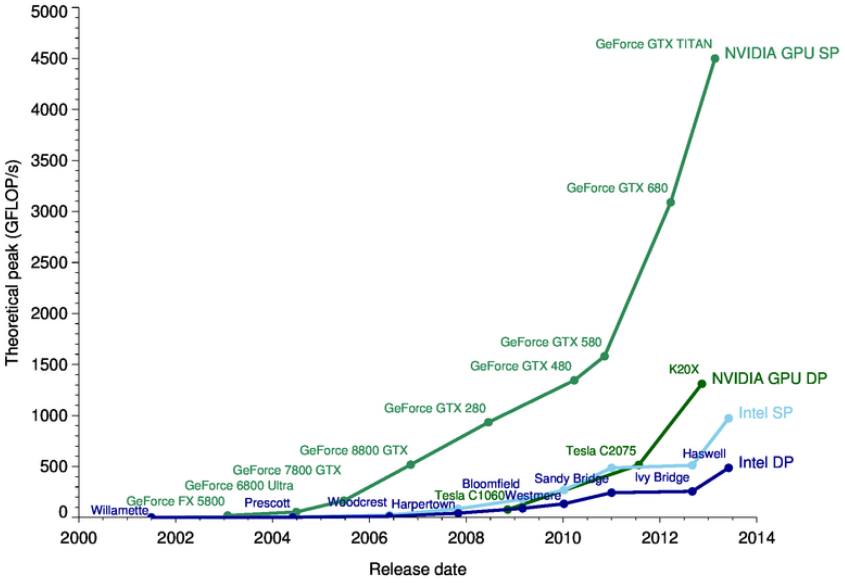


Figure 1. Peak performance for recent hardware [23].

the scientific community, but currently more and more consumer applications are being adapted to the GPU [36]. This trend became obvious after the launch of higher-productivity programming languages, such as CUDA and OpenCL.

The ML field is an early adopter of GPU computing technology. The first algorithms were implemented using the programmable vertex and pixel shaders [36]. This methodology was clearly only for expert programmers, as any program had to be written as if it was a rendering application. Thus, input data was loaded as texture and vertex values in GPU memory and manipulated to perform the required computation during a number of rendering passes. A good overview of some of these approaches was presented by Lopes and Ribeiro in the form of a timeline [31].

The first ML algorithm mapped to the GPU was the multi-layer perceptron (MLP) [46, 55], both due to its simplicity in implementation and because it does not have such high memory requirements as other algorithms [40, 50]. Both implementations made use of OpenGL and shaders to implement their functionality and obtained speed-ups ranging between 3 and 20 times compared to equivalent CPU implementations. These early results formed the motivation behind exploiting the GPU for other types of problems. Thus, the MLP was quickly followed by implementations of Self-Organizing Maps [9, 65], Genetic Algorithms [61, 64], Convolutional Neural Networks [14], and Belief Propagation [8, 63]. Speed-ups of more than 3-4 times compared to equivalent CPU implementations were reported in each study. After the launch of CUDA in 2007 an increasing number of scientists started making use of GPU hardware to speed up their computation.

The first GPU implementation of the SVM was actually developed using CUDA [11]. The authors reported speed-ups of 5 to 32 times for SVM training, and over 120 times for classification when compared to a LibSVM implementation running on a dual-core CPU. This was followed by several open-source SVM implementations, such as cuSVM [10], and later by GPUMLib [31] and GPU-LibSVM [3]. GPU-LibSVM [3] is a 1-to-1 mapping of the classic LibSVM, having the same interface and producing identical results. It is using the highly-optimized cuBLAS [38] library internally, thus offering significantly reduced processing time.

This trend of mapping computationally demanding algorithms to the GPU continued with parallel implementations of other complex algorithms. Classifiers such as deep neural networks [16] and complex feature extractors such as SIFT/ASIFT [18, 62] have been mapped to GPUs, with reported speed-ups of up to 70 times when compared to their CPU counterparts. These developments create

DOI: 10.1002/cpe

the possibility of having a complete feature extraction and feature classification pipeline executed entirely on the GPU.

2.2. Automatic Parallelization

Automatic or semi-automatic parallelization of sequential computer programs is the preferred method for making use of parallel hardware, mostly because the programmers are already accustomed to their own programs and prefer extending them to creating new ones. In both industrial and academic environments, most of the expertise lies in the specific scientific field of the people involved. Fields as diverse as physics, chemistry, astronomy or finance need to make use of parallel computing in order to perform high-accuracy simulations at reasonable speed. However, *parallel* programming is different to traditional, *sequential* programming. In order to fully benefit from these parallel architectures, most of the specialized hardware components need to be used, creating a *programming gap* between these HPC skills and the traditional programming skills. This gap is referred to as the ninja performance gap and is identified in [28].

Several possible solutions have been proposed for bridging this gap and hence for making parallel programming easier. The most popular are:

- Custom parallel library design.
- Automatic or semi-automatic parallelization tools.

Library implementations usually offer the best execution performance, simply because they are manually tuned for the parallel hardware they target. However, as hardware evolves, these libraries have to be adapted to reflect the evolution in hardware features. With each new hardware generation the maintenance costs are growing in order for the library user to achieve high performance levels while at the same time adhering to backward compatibility. There are currently many accelerated libraries designed for multi-core CPUs, GPUs, the many-core Intel Xeon Phi, and other accelerators. Examples of successful implementations for GPU-optimized libraries are the NVIDIA designed cuBLAS [38], cuFFT, as well as the NPP library for image processing [39]. Besides these libraries, almost every application domain features one or more toolkits that accelerate originally sequential algorithms by making use of GPU hardware. Libraries are particularly useful in cases where developers already made use of specific CPU library calls, e.g. BLAS. In this case, if library designers implement a 1-to-1 mapping of the original library to the GPU, the programmer can use it seamlessly, as is the case for cuBLAS.

Automatic parallelization tools promise to fill the programming gap described above by performing the translation steps between sequential and parallel implementations in an automatic or semi-automatic way. Issues such as loop dependencies and pointer aliasing [5] need to be handled, and hence automatic tools are often *semi-automatic*, requiring extra help from the programmer. Probably the most successful semi-automatic parallelization paradigm is OpenMP [20]. It is widely used in both research and commercial environments for harnessing the performance of multi-core processors. Parallelizing for multi-core is however simpler to achieve, as all cores share the same address space and the memory hierarchy is implicit. Parallelizing for accelerators has to overcome these barriers. Thus, semi-automatic tools for GPU parallelization usually require extra directives that describe GPU-specific aspects such as marking the arrays that should be copied, the regions that should be treated as GPU kernels and so forth. These compiler directives are given by programmers through augmenting their code through #pragma directives, as is also the case with OpenMP.

Many semi-automatic parallelization systems that target accelerators have been proposed. The most well known are based on compilers adhering to the PGI accelerator model [60] and later to the OpenACC standard [45]. Also, there are numerous initiatives for creating such tools, mostly stemming from academia. Some of the earliest ones are PIPS [26], that has evolved into Par4All [2], and the SUIF compiler [59], that were developed long before GPUs were used for general-purpose computing. Also, recent tools such as Kernelgen [34], Polly [24], hiCUDA [25], and the GPSME toolkit [57] are gaining in popularity and successful use-cases. Successful examples of semi-automatically parallelized programs range from medicine [56] to physics simulations [33].

DOI: 10.1002/cpe

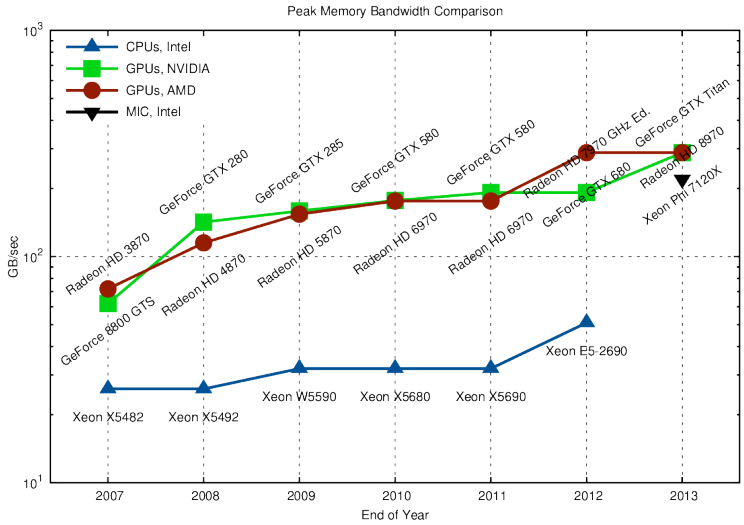


Figure 2. Peak bandwidth for recent hardware [47].

Some of the tools only make use of the raw computational power of accelerators, while other tools offer implicit or explicit support for optimizing the usage of the GPU's complex memory hierarchy and hence minimizing communication [6]. As is presented in Figure 2, the peak bandwidth of GPU devices goes beyond 200GB/s, much higher than the peak bandwidth for CPUs. However, applications requiring a high memory transfer-to-compute ratio are usually limited by memory bandwidth. Tools such as Mint [53] are alleviating this problem by using registers as much as possible, and also by automatically using shared-memory for some domain-specific patterns.

3. THE SUPPORT VECTOR MACHINE

3.1. Machine Learning

Machine learning algorithms are very useful for many applications in the field of intelligent systems such as for solving classification, regression, and adaptive control problems. These algorithms use datasets or experiences to fit a model that is able to generalize to novel, unseen examples. Therefore, after a machine learning algorithm is executed to infer a model from a dataset, it can be immediately used in the real world. Some example applications are: object recognition, face recognition, zip-code recognition, fMRI-scan classification, medical diagnosis, and document classification.

In supervised learning, the aim is to use an algorithm that creates a classification model from labelled data. The training data consist of ℓ input vectors and target labels: $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\}$, where $\mathbf{x}_i$ is an input pattern consisting of d variables, and y_i is the label belonging to that input pattern. By learning multiple models, all machine learning algorithms can deal with an arbitrary number of class labels. For example, in our experiments we will use the algorithms for handwritten digit classification in which there are 10 different class labels.

There are many different machine learning algorithms that can be used for inferring a classification model using the training data. The most often used machine learning algorithms are multi-layer perceptrons [46, 55], decision trees [44], Bayesian networks [41], k-nearest neighbors [22], and support vector machines [19, 48, 52, 54]. Because SVMs often outperform other machine learning algorithms, the SVM is currently the method of choice for dealing with many different classification problems.

DOI: 10.1002/cpe

3.2. Support Vector Machines

The support vector machine is based on structural risk minimization theory [54]. The aim is to train a model that performs well on the training data, but that also generalizes well. Let's denote the output of an SVM model when it is given an input pattern $\mathbf{x}$ with $g(\mathbf{x})$. We would like to find the function $g(\cdot)$ that is most suitable to the data.

The target labels y_i for binary classification are either 1 or -1. To explain the workings of the SVM, it is easiest to start with the linear SVM. In the linear case the SVM model is given by:

$$g(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b$$

Here $\mathbf{w}$ is the weight vector of the linear SVM, and b is the bias value. The theory of the SVM requires the output of the SVM to be always equal or larger than the target label if the label is 1, and the SVM output should be equal or lower than -1 if the label is -1. To deal with non-linearly separable data, the following soft constraint is most often used :

$$y_i(\mathbf{w}^T \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$$

Here $\xi_i \geq 0$ is called the slack variable for pattern $\mathbf{x}_i$ and the use of these slack variables allows for some errors on training examples, although it is still the goal of the learning algorithm to minimize these errors. Next to this constraint for all training examples, the idea of the SVM is to minimize the norm of $\mathbf{w}$. Therefore, the following *primal objective* should be minimized:

$$L_p(\mathbf{w}, \boldsymbol{\xi}, b) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{\ell}\xi_i$$

subject to constraints:

$$y_i(\mathbf{w}^T \cdot \mathbf{x}_i + b) \geq 1 - \xi_i$$

Here C is a meta-parameter that determines how much error the model tolerates. Therefore, it can be seen as a regularization parameter that trades off the complexity of the model and the error on the training data. The minimization of the primal objective is a convex quadratic optimization problem. Because of the convexity property there is only one minimum, which is the global optimum. However, in order to use the full power of the SVM, the optimization usually takes place using the *dual objective*. The dual-objective function is[†]:

$$L_d(\boldsymbol{\alpha}) = \sum_{i=1}^{\ell}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i \cdot \mathbf{x}_j)$$

subject to the constraints: $0 \leq \alpha_i \leq C$ and $\sum_{i=1}^{\ell}\alpha_i y_i = 0$. This dual-objective function should be maximized with respect to the α_i values.

Instead of a linear model, we can use a particular *kernel function* to replace the dot product between the patterns $\mathbf{x}_i$ and $\mathbf{x}_j$. The kernel function allows to obtain a much more powerful, non-linear classifier in a simple way. The most widely used kernel function is the radial basis function (RBF), given by:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\sum_{a=1}^{d}\frac{(\mathbf{x}_i^a - \mathbf{x}_j^a)^2}{\sigma})$$

Where $\mathbf{x}_i^a$ denotes the a^{th} input value of input pattern $\mathbf{x}_i$ and σ is a parameter that determines the kernel width. The kernel or Gram matrix is an $\ell \times \ell$ matrix containing all similarities between all training patterns. Usually the kernel matrix is computed once at the start of the SVM algorithm and the complexity of this computation is $O(\ell^2 d)$. Because of the quadratic dependence on the number of training examples, SVMs can become very slow when there is a large amount of training data.

[†]We refer to [19] for the full derivation.

DOI: 10.1002/cpe

Many SVM optimization methods exist, among which the most popular method is SMO [42]. The often used SVM library LibSVM (to which we also compare our method in this paper) also uses SMO as optimization technique. SMO uses some heuristics to change two support vector coefficients at the same time that maximize the dual objective. We developed a gradient ascent algorithm that optimizes the values of the variables α_i at the same time, which makes it easier to parallelize our method as shown later in this paper

When the gradient of the dual-objective function with respect to α_i is used, we obtain the following learning rule:

$$\alpha_i \leftarrow \alpha_i + \lambda(1 - \sum_{j=1}^{\ell} \alpha_j y_j y_i K(\mathbf{x}_i, \mathbf{x}_j))$$

where λ is the learning-rate meta-parameter. The α_i values are all updated for a specific number of epochs. After this the bias value is computed by:

$$b = \frac{\sum_{i=1}^{\ell}(y_i - g(\mathbf{x}_i)) \cdot \mathbf{1}(\alpha_i > 0) \cdot \mathbf{1}(\alpha_i < C)}{\sum \mathbf{1}(\alpha_i > 0) \cdot \mathbf{1}(\alpha_i < C)}$$

Here $\mathbf{1}(c)$ is the characteristic function that return 1 if c is true and 0 otherwise. So, in the above equation examples are only used for computing the bias if their support vector coefficients are not bounded (so larger than 0 and smaller than C).

Finally, once the SVM model is computed (which means that the support vector coefficients α_i and the bias value are optimized), it can classify new input patterns using:

$$g(\mathbf{x}) = sign(\sum_{i=1}^{\ell} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b)$$

So, for training the SVM model, we perform an initialization step, some iterations in which the support vector coefficients are optimized, and one step in which the bias value is computed. After that the model is usually tested on a dataset with unseen examples, the so-called test data. The complete training algorithm is given in Algorithm 1.

Algorithm 1 The gradient ascent SVM algorithm

Initialize α-values to a constant:
for i = 1 to ℓ **do**
  $\alpha_i = C \cdot c_1$
end for
Compute the kernel matrix for the SVM:
for i = 1 to ℓ **do**
  for j = 1 to ℓ **do**
    $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\sum_{a=1}^{d} \frac{(\mathbf{x}_i^a - \mathbf{x}_j^a)^2}{\sigma})$
  end for
end for
repeat
  Use the gradient ascent learning rule to update all α-values:
  for i = 1 to ℓ **do**
    $\alpha_i = \alpha_i + \lambda(1 - \sum_{j=1}^{\ell} \alpha_j y_j y_i K(\mathbf{x}_i, \mathbf{x}_j))$
    $\alpha_i = \text{bound}(\alpha_i)$
  end for
until maximum number of epochs is reached
Compute the bias value:
$b = \frac{\sum_{i=1}^{\ell}(y_i - g(\mathbf{x}_i)) \cdot \mathbf{1}(\alpha_i > 0) \cdot \mathbf{1}(\alpha_i < C)}{\sum \mathbf{1}(\alpha_i > 0) \cdot \mathbf{1}(\alpha_i < C)}$

DOI: 10.1002/cpe

In this algorithm $c_1 \in [0,1]$ is a meta-parameter for initializing the support vector coefficients between 0 and C, and bound(α_i) makes sure the support vector coefficients stay between 0 and C.

3.3. Tuning SVMs - Parameter search using PSO

The SVM contains a significant number of meta-parameters that need to be tuned, namely: the learning rate λ, the initialization constant c_1, the number of epochs the support vector coefficients are trained, the largest value for the support vector coefficients C, and the value for the RBF kernel width σ. For optimizing the meta-parameters we use particle swarm optimization (PSO) [27, 49]. In PSO a population of possible solutions, *particles*, is iteratively moved into a new direction. In our case the solutions are the sets of meta-parameter values. The velocity vector of the movement is calculated by using the particle's own best known position, the position of the swarm's best particle ever, and the previous velocity vector.

PSO consists of a population of size S movable particles $\mathbf{m}_i$ with $i \in (1, S)$. Each particle has a velocity vector $\mathbf{v}_i$ and a memory of the best position (in terms of achieved accuracy) $\mathbf{p}_i$ it has visited so far. Furthermore, every particle knows the best position $\mathbf{p}_{global}$ the swarm has seen so far. In every iteration step the particles and their velocities are updated so that each particle moves in the direction of its own best known position and the swarm's best known position. A detailed description of the algorithm including the update formulas can be found in Algorithm 2. The strategy parameters ω, $\psi_{individual}$ and ψ_{global} determine the influence of velocity, individual best known position, and global best known position on the particle's new position.

Algorithm 2 PSO

Initialize $\mathbf{m}_i$ and $\mathbf{v}_i$ at random
$\mathbf{p}_i = \mathbf{m}_i$, $\mathbf{p}_{global} = \arg\max_{\mathbf{m}_i}(f(\mathbf{m}_i))$
while stopping criterion not reached **do**
  for all velocities $\mathbf{v}_i$ **do**
    {update velocities}
    {$rand()$ draws random uniform values between 0 and 1}
    $\mathbf{v}_i = \omega\,\mathbf{v}_i + rand()\,\psi_{individual}\,(\mathbf{p}_i - \mathbf{m}_i) + rand()\,\psi_{global}\,(\mathbf{p}_{global} - \mathbf{m}_i)$
  end for
  for all particles $\mathbf{p}_i$ **do**
    {update positions}
    $\mathbf{m}_i = \mathbf{m}_i + \mathbf{v}_i$
  end for
  if $f(\mathbf{m}_i) < f(\mathbf{p}_i)$ **then**
    {update $\mathbf{p}_i$}
    $\mathbf{p}_i = \mathbf{m}_i$
  end if
  if $f(\mathbf{m}_i) < f(\mathbf{p}_{global})$ **then**
    $\mathbf{p}_{global} = \mathbf{m}_i$
  end if
end while

In this algorithm the function $f(\cdot)$ calls the SVM training algorithm many times and returns the average testing accuracy of different cross-validation runs.

4. DIRECTIVE-BASED IMPLEMENTATION OF THE SVM

The mapping of any algorithm to an accelerator device starts from identifying the bottlenecks in the execution using profiling tools. We have started with a single-core implementation of the gradient ascent SVM described in Section 3.2. During the profiling stage, we have identified three functions with significant weight in the computation time:

DOI: 10.1002/cpe

Algorithm 3 GPSME implementation for computing the training kernel matrix

```
#pragma GPSME copy (learn_data, toDevice, TOTAL*NR_FEATURES)
#pragma GPSME copy (kernel_train, toDevice, TOTAL*TOTAL)
#pragma GPSME for nest(2) tile(tx,ty)
for (episode = 0; episode < TOTAL; ++episode)
    for (int ad = episode; ad < TOTAL; ++ad) {
       float diff;
       float sqdiff = 0.0f;
       float output;
       for (feat = 0; feat < NR_FEATURES; ++feat) {
         diff = learn_data[episode].input[feat] -
                learn_data[ad].input[feat]
         sqdiff += diff*diff;
       }
     output = expf(-sqdiff / sigma);
     kernel_train[episode*TOTAL+ad] = output;
    }
#pragma GPSME copy (kernel_train, fromDevice, TOTAL*TOTAL)
```

- The kernel matrix computation (KMC)
- The gradient-ascent learning (GAL) iterations
- The bias computation

These functions are formally described in Section 3.2. The weight of each function in the total runtime is variable, and depends on the dataset's properties such as the number of training examples and the dimensionality of the examples. The computational weight of calculating the kernel matrix ranges from 20% on the Bangla dataset with 784 features per example to over 80% in the case of the MNIST dataset using the extended 16,464 feature space. More details on the features and datasets used are given in Section 5. Nonetheless, the cumulative weight of these three functions is over 99% for all datasets, leaving plenty of room for speed-up. If only a single function would be parallelized, even if its weight would be 80%, the maximum theoretical speed-up factor would be 5, independent of the number of processing elements used, based on Amdhal's law [1].

We consider that for offering a fair comparison between CPU and GPU implementations all cores of the CPU should be used. Thus, we insert OpenMP *#pragma for* directives in the loop structures of the three relevant functions. As illustrated in Algorithm 3, the two outermost levels of the for-loop are independent and can be safely parallelized with OpenMP. The same principle applies to the gradient ascent learning and bias computation functions, making almost the whole computationally-expensive part fairly easy to accelerate on multi-core processors. After applying the OpenMP directives to the single-core implementation, we have achieved an overall speed-up of approximately 3 times when executing on the 4-core processor described in Section 6.

The semi-automatic GPU parallelization tools operate in the same way as OpenMP. They implement a set of #pragma directives, most of them similar in nature to the OpenMP ones. Considering the host-device execution model of GPUs, the CPU orchestrates the execution and controls data transfers and kernel launches. These concepts are reflected in both OpenACC and GPSME directive sets, and are used to guide the translation. The automatic code transformations include inserting CUDA/OpenCL memory allocation/transfer commands, transforming code regions into CUDA/OpenCL kernels, and launching these kernels. These extra directives are still fairly easy to understand and use by the typical OpenMP programmer, making GPU programming as simple as OpenMP programming, at least for programs with independent for-loops, as is the case of the gradient ascent SVM training algorithm.

We have thus created three directive-based implementations for the SVM:

- an OpenMP one that serves as a baseline for the evaluation

DOI: 10.1002/cpe

- a GPSME one that is used in conjunction with the GPSME translator described in Section 4.1.
- an OpenACC one that is used in conjunction with the PGI compiler

The following subsections present several optimizations to the data storage and data access patterns, optimizations that were necessary for obtaining successful parallel implementations. However, we consider these changes much easier to perform by the typical scientist, compared to re-developing the whole code structure into a different programming language and with a different execution model in mind. Moreover, these implementations are competitive in terms of execution speed with the library-based implementations, while providing much finer control, so that experimenting with different learning algorithms or with custom kernels is possible.

4.1. The GPSME Toolkit

The GPSME toolkit is a source-to-source compiler between C/C++ and CUDA/OpenCL. It was developed in the scope of an EU FP7 project, `www.gp-sme.eu`, with the goal of helping SMEs (Small and Medium Enterprises) achieve higher value by lessening their computational problems. The GPSME toolkit builds on the previously domain-specific Mint toolkit, following its design principles. It is based on the ROSE compiler framework [43], analyzing and optimizing the AST (Abstract Syntax Tree) of the input program based on a set of #pragma directives. Some of the advantages of using the GPSME toolkit compared to other parallelization tools are the support for outputting OpenCL code (hence being able to target a wider range of hardware) as well as the high-performance and *cleaner* output code it generates. The output code has the same structure and naming as the input C/C++ code, making it easy to follow and modify for performance fine-tuning. The GPSME toolkit was previously evaluated on the Polybench standard benchmark set [57]. However, real-world problems are usually different compared to synthetic benchmarks, this being one of the motivations behind this study.

We continue with presenting some of the important optimization steps that made it possible to attain results comparable to library implementations. The optimizations are presented using the GPSME implementation for reference, but the OpenACC implementation (that we also evaluate in the experimental section) benefited from the same optimizations.

Algorithm 3 lists the original GPSME toolkit implementation for the kernel matrix computation. The *learn_data* is an array containing the *TOTAL* number of examples. Each example is described by *NR_FEATURES* features. Several experiments with varying number of data elements and with a varying number of features are provided in Section 6.

Algorithm 4 AoS vs. SoA data structure

```
struct DATA{                              struct DATA{
  int class;                               int class[TOTAL];
  float input[NR_FEATURES];                float input[TOTAL*NR_FEATURES];
}                                         }

DATA *learn_data= new DATA[TOTAL];   DATA *learn_data= new DATA;
```

However, after parallelizing the code from Algorithm 3, the resulting GPU implementation performed very poorly, being slower than the CPU version. This happens because the input data is stored as Arrays of Structures (AoS), rather than Structures of Arrays (SoA). This is a basic optimization technique when using GPU hardware, and is explained by the code snippet in Algorithm 4. By applying this transformation to the data structure, the code calculating the kernel training matrix has to be slightly transformed.

This minor change in the access pattern has significant benefits on GPU architectures, the foremost being that it leads to higher memory bandwidth. By analyzing the code from Algorithm 3, we observe that the arithmetic intensity of the kernel matrix computation stage is below 1 operation/byte transferred, making the kernel matrix computation stage memory-bound on GPU architectures [58]. Hence, the translated version achieved an approximately 10-fold performance

DOI: 10.1002/cpe

Algorithm 5 Optimized GPSME implementation for computing the training kernel matrix

```
#pragma GPSME copy (learn_data, toDevice, TOTAL*NR_FEATURES)
#pragma GPSME copy (kernel_train, toDevice, TOTAL*TOTAL)
#pragma GPSME for nest(2) tile(tx,ty)
for (episode = 0; episode < TOTAL; ++episode)
    for (int ad = episode; ad < TOTAL; ++ad) {
       float diff;
       float sqdiff = 0.0f;
       float output;
       for (feat = 0; feat < NR_FEATURES; ++feat) {
       diff = learn_data.input[episode*NR_FEATURES+feat] -
              learn_data.input[ad*NR_FEATURES+feat]
       sqdiff += diff*diff;
       }
     output = expf(-sqdiff / sigma);
     kernel_train[episode*TOTAL+ad] = output;
    }
#pragma GPSME copy (kernel_train, fromDevice, TOTAL*TOTAL)
```

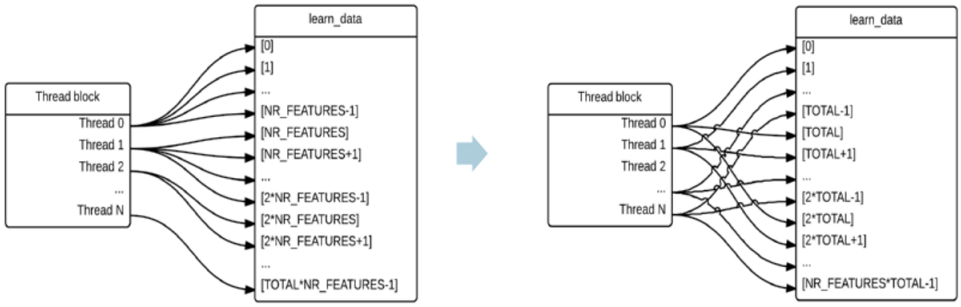


Figure 3. Transformation from uncoalesced to coalesced memory accesses.

increase when using SoA instead of AoS for representing the input data. The benefits are seen in both parallelizing tools' cases. The code resulting from this change is presented in Algorithm 5.

With both GPSME and OpenACC, the two independent outer loops can be safely parallelized onto a 2D thread block. Therefore, each device thread should run the innermost for-loop sequentially. This leads to the memory access pattern on the left side of Figure 3. In this case, the first thread reads the first chunk of *NR_FEATURES* elements from the *learn_data* array, the second thread needs the second chunk of *NR_FEATURES* elements, and so on. The recommended way is to have consecutive threads read consecutive memory locations, referred to as coalesced memory accesses. This can be achieved through reordering of the *learn_data* array. By transposing the matrix represented by the *learn_data* array (i.e. by storing the matrix in column-major order instead of row-major order) at its initialization, the code from Algorithm 5 can easily be modified to access the array in a coalesced way, adhering to the right side of Figure 3. This change made the resulting code up to 3 times faster than the previous version.

We have described the type of modifications that were necessary in order to obtain high-quality GPU code from both the GPSME toolkit and PGI's OpenACC compiler. Although the techniques presented here are exemplified using the GPSME model, the same techniques have been applied for the OpenACC model. We consider these changes very easy-to-make by the typical OpenMP programmer, if he/she is provided with some recommendations for efficient GPU computing and with a series of examples.

DOI: 10.1002/cpe

4.2. OpenACC

The OpenACC standard is currently best supported by the PGI compiler, recently acquired by NVIDIA. Similarly to the GPSME toolkit, it works internally as a source-to-source compiler from C/C++ to CUDA. Even though the output code is usually hidden from the programmer, it can be extracted through compiler flags. However, the output code is unreadable when compared to the GPSME output code that maintains the same structure, and a GPU programmer is even able to further fine-tune it. The CUDA code is then compiled by NVIDIA's NVCC compiler that comes bundled with the PGI compiler. However, the translation and compilation is usually performed in a single-step fashion, abstracting the source-to-source details from the user.

OpenACC is superior in terms of flexibility offering an extensive set of directives. Hence, for more complicated loop structures OpenACC has more options for expressing parallelism and for handling dependencies. The directives used for the OpenACC implementation of the SVM are omitted here, as they are very similar to the GPSME ones.

One of the key differences between the two semi-automatically generated implementations is that the PGI Compiler automatically detects that a reduction optimization for the innermost for-loop in Algorithms 3-5 can be performed, and instantiates a sum-reduction kernel. The GPSME simply executes the innermost for-loop on each device thread independently. Another important difference is that the code generated by the GPSME toolkit makes better use of registers, trying to minimize accesses to the global memory. In contrast, OpenACC performs global memory accesses when reading/writing most of the variables. Hence, as demonstrated in Section 6, the code outputted by the GPSME toolkit is consistently faster than the one generated by the PGI compiler.

5. HANDWRITTEN DIGIT CLASSIFICATION

Handwritten digit classification aims to infer the ground-truth labels of the provided handwritten images. Recognizing handwritten digits is very important for automatic zip-code recognition and it also has applications in analyzing handwritten documents. Compared to optical character recognition, in which it is the aim to infer the character from an image displaying a typed letter, handwritten digit classification is much harder. The reasons are that there are many different human writers using their own writing style and therefore the variations in the images are much larger.

A large number of studies investigated the problem of handwriting recognition based on the MNIST dataset [30]. The MNIST dataset was modified from the original NIST database [30], and is nowadays used as a standard benchmark for testing machine learning techniques and pattern recognition methods [15, 32]. In the MNIST dataset, there are $60,000$ handwritten digit images for training and $10,000$ test images. This setup of one single fixed test dataset allows for fair comparisons between algorithms, but in this paper we will create multiple splits using a smaller training dataset. The size of these handwritten digit images is normalized and the digits are centered in a fixed-size image to fit into a 28×28 pixel space [30, 32]. Furthermore, the handwritten images are completely separated from the background. Although this is a large dataset, most digits are clearly written and the dataset is researched extensively. In Figure 4 we display a number of images for the digits 0-9 from the MNIST dataset. The highest accuracy on this dataset was obtained in [17], where an accuracy of 99.73% was obtained.

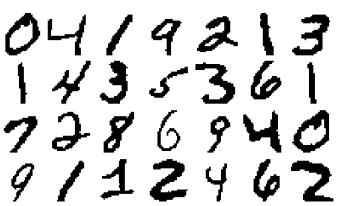
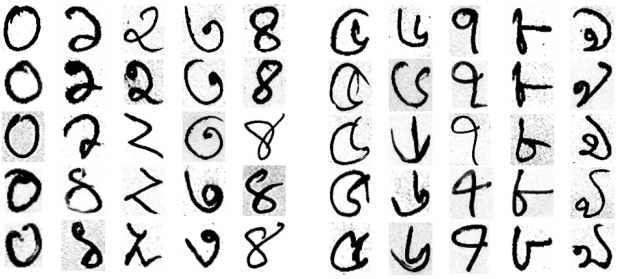


Figure 4. Some examples taken from the MNIST dataset.

DOI: 10.1002/cpe

Next to using the MNIST dataset in our experiments, we will also focus on the recognition of handwritten *Bangla* (or Bengali) digits, which is the second most popular language in India and Bangladesh [21]. The Bangla dataset is composed of $10,920$ digit examples. Out of these examples, we randomly select 90% of them as being training examples, and 10% as test examples. Hence, each evaluated SVM is trained on 9828 examples and tested on 1092 examples. The dataset contains different kinds of background and a variety of pixel space resolutions. Various example digits of handwritten Bangla are shown in Fig. 5. It is clear that, when compared to the MNIST dataset, Bangla digits are more complicated and there is more style diversity [7]. For instance, the curly tails in Bangla characters makes the definition of a stable bounding box problematic. The best previous results on this dataset were obtained with an ensemble machine learning technique, where an accuracy of 96.8% was obtained [51].



(a) (b)

Figure 5. A variety of handwritten Bangla digit samples. (a) Set of numbers from 0 to 4 and (b) from 5 to 9. Note the large differences between the examples on the first and last rows.

5.1. Feature extraction methods

In handwritten digit recognition it is possible to immediately use the pixel intensities from the gray images as input features for the machine learning algorithm. However, in some cases better results can be obtained if there is a feature extraction algorithm that first converts the gray image to a set of features, after which these extracted features become the input for the learning algorithm.

In this paper we will compare using the pixel intensity information to using two different feature extraction methods that convert the images to a high-dimensional input vector.

Using pixel intensities. The direct use of pixel intensity information is a straightforward method to convert a 2D image to an input vector. Because the dataset with images is stored using pixel values between 0 and 255, all we need to do is to normalize the inputs so they will fall in the range between -1 and 1. The MNIST and Bangla datasets both use character images of size 28×28, therefore the use of pixel intensities results in 784 input features.

Oriented line segment products. In machine learning there has been a lot of evidence that increasing the dimensionality of the input vectors can improve recognition performance. Therefore, instead of only using the 784 pixel intensities, this method adds horizontal, vertical, and diagonal line information using line segments of different sizes. In Figure 6 this idea is illustrated. The figure shows two pixels in which line segments of length 2 and length 3 are used to extract multiple inputs. In the product algorithm the pixel intensities are first converted to lie between 0 and 1, where a value of 1 stands for completely black. After this, the oriented line segment product feature extraction algorithm computes the product of all pixel intensities lying on a line segment of a particular length. In the end, this method uses the pixel intensity itself and adds as additional inputs for each pixel the line segment product in 4 directions of lengths 1 until 5. Therefore this method results in $784 + 4 \times 5 \times 784 = 16,464$ features in total. Because line segments can pass the border of an image, we pad the contours of the image with pixel intensities of 1 to ensure the product returns a well-defined value. After computing the product features, all inputs were normalized to fall in the range between -1 and 1.

DOI: 10.1002/cpe

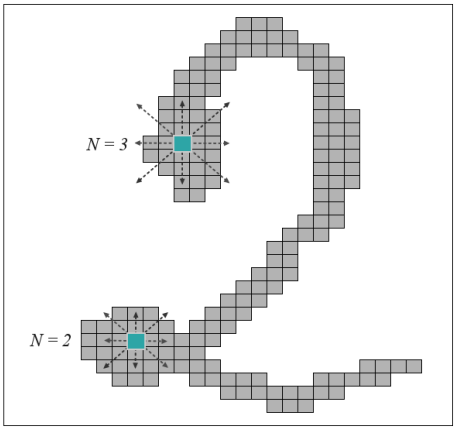


Figure 6. Illustration of line-segment features. In this image, from each pixel multiple inputs are generated by using vertical, horizontal, and diagonal line features constructed from different lengths of line segments. In the image the use of line segments of lengths 2 and 3 are shown.

Oriented line segment sums. This feature extraction method is very similar to the oriented line segment product feature extraction method explained above. The main difference is that pixel intensities on line segments are summed up to compute a value for a particular orientation and length of a line feature. Furthermore, padding is now done using zero-values. After summing the pixel intensities on line segments all sums are renormalized to fall in the range between -1 and 1. We use the same setup as with the line product so we use four orientations and 5 different line segment lengths, and therefore this feature extraction method also results in $784 + 4 \times 5 \times 784 = 16,464$ input features.

6. EXPERIMENTAL RESULTS

In order to have a fair comparison of current SVM classification systems for handwriting recognition, we present results in terms of both execution performance and classification accuracy. The task chosen is handwritten digit classification using the Bangla and MNIST datasets described in Section 5.

In machine learning it is a common approach to use cross validation in which the whole dataset is randomly partitioned a number of times in training data and test data. Then the training algorithm is repeatedly invoked on the training data and tested on the test data to obtain the testing accuracy. By using multiple cross-validation runs the average accuracy and the standard deviation can be computed as the final result. In our experiments, we used 50 cross-validation runs.

The baseline implementation is the gradient-ascent based SVM augmented with OpenMP directives for using 4 CPUs in parallel. Along with it, we evaluate a CPU LibSVM implementation [13] and its OpenMP version, a high-performance GPU LibSVM implementation [3] based on the highly-optimized cuBLAS library, and two semi-automatic parallelization tools operating on the baseline implementation as described in Section 4, that is the industry-standard PGI OpenACC compiler and the GPSME toolkit. Moreover, considering the LibSVM implementation, we have added OpenMP *#pragmas* and recompiled it following the guidelines on the authors' website. This gives a deeper insight into how the performance of the library approach scales with the number of CPU cores.

The test machine is comprised of a quad-core Intel Core i7-3770K 3.5 GHz CPU and an NVidia GTX690 GPU. Despite the fact that the GTX690 board is a dual-GPU board, we are using a single GPU for all experiments presented. The tests were performed under Ubuntu 12.04 LTS using GCC 4.6 as the C/OpenMP compiler, NVCC (from the CUDA 5.0 SDK) as the CUDA compiler, and PGI 13.1 as the OpenACC compiler. All tests were compiled using maximum optimization settings.

DOI: 10.1002/cpe

For the two semi-automatically parallelized versions we have exhaustively chosen the best thread block/grid size combinations in order to have the fastest performing output code. The PSO parameter search described in Algorithm 2 was used in order to find the best parameters for these two semi-automatically generated implementations, as well as for the baseline OpenMP implementation. The parameters that need to be tuned are explained in Section 3.2. The GPSME-generated implementation was used for parameter search, as it was the fastest option.

LibSVM also includes a utility to automate parameter searches, but that method does not perform as well as the PSO. Since it uses the Sequential Minimal Optimization (SMO) training strategy, some other parameters need to be tuned. In order to make the comparison as fair as possible we use the included utility to find the optimal parameters. The parameters found for LibSVM are also used to evaluate GPU-LibSVM, since it is a 1-to-1 mapping of LibSVM.

6.1. Pixel-feature comparison

In this subsection we present the execution times and accuracy results for all SVM implementations evaluated using the *traditional* pixel-based feature, as described in Section 5.

Table I presents the timing and accuracy results for classifying the Bangla dataset with the best parameters found. The speed-up offered by the semi-automatic parallelization tools, especially the GPSME one, is impressive, offering an overall speed-up of 6.7 times compared to the baseline OpenMP version. It is also 2.5 times faster than the original LibSVM implementation and achieves 43% of the highly optimized GPU-LibSVM performance. The OpenACC implementation is about 2 times slower than the one obtained using the GPSME toolkit. The GPU-LibSVM version is 1.8 times faster than its CPU counterpart, the OpenMP LibSVM. This makes the speed-up for our GPU-SVM implementation compared to the baseline CPU one much higher than the one of GPU-LibSVM compared to its CPU counterpart.

Table I. Evaluation of the classification results for the Bangla dataset using the pixel-based features

SVM implementation	Accuracy [%]	Standard error [%]	Execution time [s]
Baseline OpenMP	97.34	0.45	161.2
LibSVM	96.70	n/a	60.5
LibSVM OpenMP	96.70	n/a	19.2
GPU-LibSVM	96.70	n/a	10.5
PGI OpenACC	97.34	0.45	45.8
GPSME	97.34	0.45	24.1

After evaluating the Bangla dataset we use the same implementations described above for classifying the handwritten digits from the MNIST dataset. The MNIST dataset is composed of $60,000$ training examples and $10,000$ test examples. Out of these examples we randomly extract $10,000$ training and $5,000$ test examples, simply because the GPU device does not have enough memory to store the kernel matrix. The kernel matrix grows quadratically with the number of examples, so for running the current GPU SVM classifier on the full MNIST dataset it needs $60,000 \times 60,0000 \times 4bytes = 13.41$GB of storage space on the GPU. This is a limitation of all evaluated GPU implementations. One simple solution for overcoming this problem is forcing the GPSME toolkit to use the *cudaMallocHost* function to allocate pinned memory when going past the memory capacity of the GPU. The data that are not on the GPU are cached from the global DRAM when needed, limiting the achievable speed-ups. However, these experiments with larger datasets are beyond the scope of this paper.

Table II presents the results for the MNIST dataset as described above. The results of the semi-automatic parallelization on MNIST are similar in terms of relative performance to the ones attained on Bangla. The GPSME-accelerated version is 6.4 times faster than the baseline and 2 times faster than the PGI version. It is also reaching about 40% of the performance of the library approach.

The OpenMP version of LibSVM scales well in terms of number of cores, being about 3 times faster than the single core LibSVM on both datasets when using 784 features per example. This

DOI: 10.1002/cpe

speed-up is expected, the baseline OpenMP implementation seeing a similar one as stated in section 4. This comes from the fact that examples are rather small, $3,136$ bytes each, and so up to 10 examples can be cached in the L1 data cache of the CPU at any time. The LibSVM implementation using OpenMP is much faster than the baseline OpenMP one mostly because of the different learning strategy it uses. However, as outlined in the next section, the execution performance of the SMO-based method is heavily impacted by the dimensionality of the feature space, much more than the gradient ascent-based one is.

Table II. Evaluation of the classification results for the MNIST dataset using the pixel-based features

SVM implementation	Accuracy [%]	Standard error [%]	Execution time [s]
Baseline OpenMP	97.65	0.18	130.3
LibSVM	97.17	n/a	45.0
LibSVM OpenMP	97.17	n/a	13.4
GPU-LibSVM	97.17	n/a	7.8
PGI OpenACC	97.65	0.18	41.4
GPSME	97.65	0.18	20.4

6.2. *Extended feature comparison*

The memory limitation described above is only related to the dataset size in terms of the number of examples. In terms of input feature dimensionality, the memory requirements grow linearly. The kernel matrix grows quadratically with the number of examples, as explained before, but is independent of the input dimensionality. The input data grows linearly in terms of number of examples and also in terms of the data dimensionality. For example, by extending the dimensionality of the example from 784 to $16,464$ as described in Section 5, the memory requirements of the input data buffer is still manageable. The memory requirement for the extended feature is $10,000 \times 16,464 \times 4bytes = 628$MB of data, compared to about 30MB for the pixel feature. This technique can be applied on datasets with very large input dimensionality, because the acceleration potential is even higher in this case.

The results on both the Bangla and MNIST datasets using the extended feature are presented in Tables III and IV. With these features, the GPSME version became the fastest of all implementations. The speed-up of the GPSME version is between 8 and 10 times compared to the baseline OpenMP version, slightly higher than the results for the reduced feature space. However, the difference in performance between the LibSVM CPU implementation and the gradient ascent SVM CPU implementation is smaller when the feature space is extended. The baseline OpenMP implementation is even faster than the LibSVM OpenMP implementation when classifying the Bangla examples. This leads to the conclusion that SMO-based methods tend to get much slower than gradient ascent ones do when increasing the feature space. These differences arise from the different learning strategies employed by the two approaches. It also explains why our method is even faster than the GPU library-based one on the Bangla dataset, GPU-LibSVM relying also on SMO. In the case of the gradient-ascent SVM, the timing results on these extended feature

Table III. Evaluation of the classification results for the Bangla dataset using the extended features

SVM implementation	Feature	Accuracy [%]	Standard error	Execution time [s]
Baseline OpenMP	Sum	97.53	0.41	535.4
LibSVM	Sum	97.12	n/a	1334.8
LibSVM OpenMP	Sum	97.12	n/a	789.3
GPU-LibSVM	Sum	97.12	n/a	126.7
PGI OpenACC	Sum	97.53	0.41	105.1
GPSME	Sum	97.53	0.41	51.3
GPSME	Product	97.51	0.52	45.5

DOI: 10.1002/cpe

Table IV. Evaluation of the classification results for the MNIST dataset using the extended features

SVM implementation	Feature	Accuracy [%]	Standard error [%]	Execution time [s]
Baseline OpenMP	Sum	97.52	0.20	709.5
LibSVM	Sum	97.16	n/a	906.1
LibSVM OpenMP	Sum	97.16	n/a	536.1
GPU-LibSVM	Sum	97.16	n/a	84.5
PGI OpenACC	Sum	97.52	0.20	159.2
GPSME	Sum	97.52	0.20	89.3
GPSME	Product	98.18	0.17	84.1

spaces are dominated by the KMC kernel. For MNIST, the KMC kernel computes two matrices of size $10,000 \times 10,000$ and $5,000 \times 5,000$, while for Bangla it computes smaller matrices of size $9,828 \times 9,828$ and $1,092 \times 1,092$. This explains why the classification on MNIST is consistently slower than on Bangla on all approaches that use gradient-ascent as learning strategy. As with the reduced feature, the OpenACC version is about half as fast as the GPSME one, but still much faster than the CPU implementations.

The GPU library version, GPU-LibSVM is clearly surpassed by the automatically parallelized versions on the Bangla dataset, while having comparable results to the GPSME version on MNIST. As also observed for the CPU implementation comparison, the reason behind this is that the SMO-based methods tend to get much slower than gradient ascent ones do when increasing the feature space.

Another interesting observation is that the performance of the LibSVM OpenMP version does not scale particularly well at this large feature space. The example size in this case is about 64KB, each core being forced to swap the data cache for each example. This effectively lowers the memory bandwidth of the application, limiting the speed-up achieved by OpenMP to below 1.7 times. Hence, using GPUs for large feature spaces is clearly the method of choice, the speed difference between CPU and GPU implementations being highly significant for these cases.

To clearly show how the GPSME version scales in terms of both number of examples and feature dimensionality, we have performed a final experiment in which we have varied both parameters for the classification of MNIST. Hence, we have experimented with the traditional 784-dimensional features, the extended $16,464$-dimensional features, as well as medium-sized features, $10,192$-dimensional, extracted using a maximum line segment length of 3, as described in Section 5. Moreover, when choosing the extended feature, we also vary the number of train/test examples. To better reason on the results, we present per-function execution timings in Table V. The functions profiled are the initialization, KMC, and the GAL functions, as presented in Algorithm 1. The time remaining up to the total execution time is spent in the compute bias kernel. The conclusion for this experiment is that the KMC kernel scales almost linearly with the size of the feature space, and almost quadratically with the number of examples. This is expected, reflecting the algorithmic complexity of Algorithm 5. As feature dimensionality increases, the weight of KMC in the total execution time is also increased. The GAL kernel is not that predictable, it's execution time being based on the parameters found in the parameter tuning stage, such as the number of learning iterations. One interesting observation is that in the case of MNIST, the full set of train and test examples is loaded at runtime and a specific part is randomly selected for the experiment. Thus, lowering the number of train and test examples to $2,500$ and $1,250$ respectively, keeps the execution time of the initialization part mostly unchanged. As a result, our experiments conclude that increasing the feature size by 21 times increases the total execution time only 4-fold for the case of MNIST.

6.3. Discussion

Apart from the execution performance comparison discussed above, a very interesting aspect is the accuracy attained by the gradient ascent based implementations. The PSO algorithm consistently

DOI: 10.1002/cpe

Table V. Scaling the GPSME approach for the MNIST dataset

Feature Size	Nr. examples	Initialization [s]	KMC time [s]	GAL time [s]	Total time [s]
784	10000/5000	4.3	2.7	12.8	20.4
10192	10000/5000	19.9	23.4	22.4	66.7
16464	10000/5000	31.5	37.5	14.3	84.1
16464	5000/2500	30.7	10.4	4.0	46.0
16464	2500/1250	29.9	3.2	1.1	35.3

finds better parameters for the gradient ascent SVM than the parameter searcher for the LibSVM family of classifiers. Actually, we obtain the best accuracy result of 97.5% for the Bangla dataset, which is significantly better than the previous best results of 96.8% [51]. The result is stable, being obtained after 50-fold cross-validation runs.

This has been possible through extensive parameter searches. If we take the MNIST case with the extended features, running 2000 instances of the baseline gradient ascent SVM for parameter tuning would take more than 2 weeks. If someone would implement the same PSO algorithm for the standard LibSVM, it would take more than one month for the 2000 instances. We could manage to run these 2000 experiments in less than 2 days using the semi-automatically generated GPSME version. The only effort required from the scientist to achieve this is to (re-)organize the C/C++ code in a more GPU-friendly way, and to add some extra directives to it. The performance of the OpenACC-accelerated version, although about half of GPSME's, is still much better than of the CPU versions when classifying highly-dimensional datasets. The speed difference between the GPSME and OpenACC version is mostly explained by two facts. Firstly, the GPSME toolkit makes extensive use of the register file when accessing thread-private variables, as opposed to OpenACC that is mostly using global memory for this. This phenomenon was also presented for the case of micro-benchmarks in [57]. Secondly, as explained in Section 4.2, the PGI compiler automatically detects that a reduction operation is possible in the innermost for-loop and hence infers a sum-reduction kernel that may not always be profitable.

It is interesting to also analyze the performance of the GPSME-based in comparison with the GPU-LibSVM library-based classifiers. When increasing the feature space, the relative difference between these implementations shrinks, and for the Bangla scenario with an extended feature space the generated implementation is actually faster than the library one. We see two reasons for this. Firstly, as also noted previously, the two SVM implementations use a different learning strategy, the gradient ascent learning strategy proving to be faster for highly-dimensional feature spaces. Secondly, when using the reduced feature space, the memory hierarchy is used more efficiently by the library version, as entire examples can be cached. When using the extended feature, because each example is larger than the whole cache/shared memory of a GPU Streaming Multiprocessor, the gains from using the library are diminished, as accesses are performed anyway from/to global memory. To achieve performance close to the library version even for small examples, the GPSME toolkit has to be enhanced with the ability of detecting parts of code that can be automatically replaced by CUBLAS calls. However, this is beyond the scope of this paper.

If we examine the results of the extended features, then we can observe that the sum feature leads to significantly better results than the pixel intensities on the Bangla dataset, Furthermore, on the MNIST dataset the product feature obtains much higher accuracies than using pixel intensities directly. This difference is also highly significant. Without the GPSME implementation it would have been very time-consuming to obtain the best parameters and the final results of the extended features, but using our highly optimized GPU-SVM program, working with high-dimensional feature spaces becomes an interesting possibility. Hence, one important conclusion is that the SVM version generated with the GPSME toolkit is very appropriate to use in cases where the feature dimensionality is very high and the number of examples is moderate. A larger number of examples that surpass the memory capacity can also be supported, by for example using cudaMallocHost as explained previously, but this is beyond the scope of this paper.

DOI: 10.1002/cpe

7. CONCLUSION

Semi-automatic parallelization tools enable faster development of code targeting parallel systems. This model is accepted, as it has already been applied in OpenMP and has been adopted by both academia and the industry. By following some key principles such as data organization and accesses, and adding some OpenMP-like directives, the scientist can achieve major speed-ups.

We have evaluated several implementations of a very important Machine Learning algorithm, the Support Vector Machine. CPU, GPU library, and semi-automatically-generated implementations were evaluated on the task of classifying digits from the MNIST and Bangla handwritten digit datasets. Especially when faced with large feature spaces, the GPSME-augmented version performed the fastest. Extensive parameter search was possible and this led to obtaining 'the best' results for the Bangla dataset.

As future directions we will apply this technique to datasets with even larger feature spaces, such as the ones present in biology in which micro-array data is used to recognize inborn diseases or in neuroscience in which fMRI scans lead to around 500,000 voxel values that serve as input to recognize for example degenerative diseases. Another direction for future work is creating a pre-processing tool for the GPSME toolkit that gives code modification recommendations to the application programmer (e.g. what to change to have coalesced memory accesses), in order to enable even faster development of high-performance code. Finally, considering that the GPSME toolkit can generate OpenCL output code that can be executed by both the CPU and GPU, future work will also explore the generation of heterogeneous CPU-GPU programs with the use of the toolkit.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Union's Seventh Framework Programme managed by REA-Research Executive Agency http://ec.europa.eu/research/rea (FP7/2007-2013) under grant agreement no. 286545. Project website http://www.gp-sme.eu.

REFERENCES

1. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.
2. M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon, et al. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.
3. A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris. GPU acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, 2011.
4. S. Baboo, P. R. Kumar, et al. Next generation data warehouse design with big data for big analytics and better insights. *Global Journal of Computer Science and Technology*, 13(7), 2013.
5. U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.
6. M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.
7. T. Bhowmik, P. Ghanty, A. Roy, and S. Parui. SVM-based hierarchical architectures for handwritten Bangla character recognition. *International Journal on Document Analysis and Recognition*, 12:97–108, 2009.
8. A. Brunton, C. Shu, and G. Roth. Belief propagation on the GPU for stereo vision. In *Computer and Robot Vision, 2006. The 3rd Canadian Conference on*, pages 76–76. IEEE, 2006.
9. A. Campbell, E. Berglund, and A. Streit. Graphics hardware implementation of the parameter-less self-organising map. In *Intelligent Data Engineering and Automated Learning-IDEAL 2005*, pages 343–350. Springer, 2005.
10. A. Carpenter. cuSVM: A CUDA implementation of support vector classification and regression. *patternsonscreen. net/cuSVMDesc. pdf*, 2009.
11. B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.
12. J. M. Cavanagh, T. E. Potok, and X. Cui. Parallel latent semantic analysis using a graphics processing unit. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2505–2510. ACM, 2009.
13. C.-C. Chang and C.-J. Lin. libSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
14. K. Chellapilla, S. Puri, P. Simard, et al. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

DOI: 10.1002/cpe

15. D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, 2010.
16. D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Handwritten digit recognition with a committee of deep neural nets on GPUs. *arXiv preprint arXiv:1103.4487*, 2011.
17. D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012.
18. V. Codreanu, F. Dong, B. Liu, J. B. Roerdink, D. Williams, P. Yang, and B. Yasar. GPU-ASIFT: A fast fully affine-invariant feature extraction algorithm. In *Proceedings of the International Conference High Performance Computing and Simulation*, pages 474–481. IEEE, 2013.
19. N. Cristianini and J. Shawe-Taylor. *Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.
20. L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
21. N. Das, B. Das, R. Sarkar, S. Basu, M. Kundu, and M. Nasipuri. Handwritten Bangla basic and compound character recognition using MLP and SVM classifier. *Journal of Computing*, 2(2), 2010.
22. R. Duda and P. Hart. *Pattern classification and scene analysis*. New York: John Wiley and Sons, 1973.
23. M. Galloy. CPU vs. GPU performance. http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html. Accessed: 2014-05-26.
24. T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.
25. T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.
26. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM, 1991.
27. J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.
28. C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Closing the ninja performance gap through traditional programming and compiler technology. Technical report, Technical report, Intel Labs, 2011.
29. D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.
30. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86(11), pages 2278–2324, 1998.
31. N. Lopes and B. Ribeiro. GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications ISSN*, pages 2150–7988, 2010.
32. U. Meier, D. Ciresan, L. Gambardella, and J. Schmidhuber. Better digit recognition with a committee of simple neural nets. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1250–254, 2011.
33. M. A. Mikalsen. OpenACC-based snow simulation, 2013.
34. D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström. KernelGen–the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. Technical report, USI Technical Report Series in Informatics (July 2013), 1–14, 2013.
35. M. Muja and D. G. Lowe. FLANN, fast library for approximate nearest neighbors, 2009.
36. J. Nickolls and W. J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.
37. S. Nissen. Implementation of a fast artificial neural network library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31, 2003.
38. C. Nvidia. CUBLAS library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.
39. N. P. P. NVIDIA. February 2011, 11.
40. K.-S. Oh and K. Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.
41. J. Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288, 1986.
42. J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.
43. D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.
44. J. Quinlan. *C4.5 Programs for machine learning*. San Mateo, CA: Morgan Kaufmann., 1993.
45. R. Reyes, I. López, J. Fumero, and F. de Sande. A comparative study of openacc implementations. *Jornadas Sarteco*, 2012.
46. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.
47. K. Rupp. CPU, GPU and MIC hardware characteristics over time. http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/. Accessed: 2014-05-26.
48. B. Schölkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2002.
49. Y. Shi and R. Eberhart. Parameter selection in particle swarm optimization. In *Evolutionary Programming VII*, pages 591–600. Springer, 1998.
50. D. Steinkraus, I. Buck, and P. Simard. Using GPUs for machine learning algorithms. In *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, pages 1115–1120. IEEE, 2005.
51. O. Surinta, L. Schomaker, and M. Wiering. A comparison of feature and pixel-based methods for recognizing handwritten bangla digits. In *Proceedings of the Twelfth International Conference on Document Analysis and Recognition (ICDAR)*, 2013.

DOI: 10.1002/cpe

52. J. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Processing Letters*, 9:293–300, 1999.
53. D. Unat, X. Cai, and S. B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.
54. V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1995.
55. P. J. Werbos. Advanced forecasting methods for global crisis warning and models of intelligence. In *General Systems*, volume XXII, pages 25–38, 1977.
56. S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC - First experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.
57. D. Williams, V. Codreanu, P. Yang, B. Liu, F. Dong, B. Yasar, B. Mahdian, A. Chiarini, X. Zhao, and J. B. Roerdink. Evaluation of autoparallelization toolkits for commodity graphics hardware. In *Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics*, 2013.
58. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
59. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.
60. M. Wolfe. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.
61. M.-L. Wong, T.-T. Wong, and K.-L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293. IEEE, 2005.
62. C. Wu. SiftGPU manual. http://cs.unc.edu/~ccwu.
63. Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nister. Real-time global stereo matching using hierarchical belief propagation. In *BMVC*, volume 6, pages 989–998, 2006.
64. Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation*, pages 1051–1059. Springer, 2005.
65. L. Zhongwen, L. Hongzhi, Y. Zhengping, and W. Xincai. Self-organizing maps computing on graphic process unit, 2005.

DOI: 10.1002/cpe